Java Interfaces

• Interfaces declare *features (i.e. methods)* but provide *no* implementation

• classes that implement an interface *must* provide an implementation for each feature

• Interfaces can inherit from another interface
  - at most one superinterface

• A Java class can implement more than one interface
  - Java inherits(extends) from at most one type, but can implement more than one interface.
Java Interfaces (cont)

- Printable defines 3 methods
- Displayable inherits from Printable and adds some more method signatures
- A class that implements Displayable will have to provide an implementation for all 6 methods.

```java
interface Printable {
    public String printInstVar();
    public String showInfo();
    public String printWithSpaces();
}

interface Displayable extends Printable {
    public String display();
    public String refresh();
    public void displayColor(Color c);
}
```
Java Interfaces (cont)

- **keyword** `implements` followed by a list of one or more comma separated interface names

- Method signatures must match
  - same modifiers
  - same method names
  - same number of arguments
  - corresponding argument types.

```java
public class Circle implements Displayable {
    private int radius;
    public String printInstVar() {
        System.out.println("Radius "+radius);
    }
    public String showInfo() {
        System.out.println(" I am a Triangle with radius "+radius);
    }
    public String printWithSpaces() {
        ...
    }
    public String display() {
        ...
    }
    public String refresh() {
        ...
    }
    public void displayColor(Color c) {
        ...
    }
}
```
Types Revisited

• In Java each interface defines a type. Interface extension and implementation as *subtype* relationships

• A subtype relation in Java is:
  
  – if class $C_1$ extends class $C_2$ then $C_1$ is a subtype of $C_2$
  
  – if interface $I_1$ extends $I$ then $I$ is a subtype of $I$
  
  – if class $C$ implements interface $I$ then $C$ is a subtype of $I$
  
  – for every interface $I$, $I$ is a subtype of Object
  
  – for every type $T$, $T[ ]$ is a subtype of Object
  
  – if $T_1$ is a subtype of $T_2$ then $T_1[ ]$ is a subtype of $T_2[ ]$
Types of Circle

- Circle is a subtype of
  - Object
  - Displayable
  - Printable

```java
public class Circle implements Displayable {
    private int radius;
    public String printInstVar(){
        System.out.println("Radius "+radius);
    }
    public String showInfo(){
        System.out.println(" I am a Triangle with radius "+ radius);
    }
    public String printWithSpaces(){
        ...
    }
    public String display(){
        ...
    }
    public String refresh(){
        ...
    }
    public void displayColor(Color c){
        ...
    }
}
```
Inheritance and its forms

• Combination
  – child class inherits features from more than one parent
    • Java does not *directly* support this last form, although we can simulate it (more on this next time)

• Using interfaces a class can inherit features from more than one parent.
  – parents do not have to be in an direct inheritance relationship
Students, TAs and Professors

• Modeling a department with
  - students
  - TA
  - professors

• Students
  - gpa, full-time or part-time, courses

• TA
  - is a student, office hours, course TAing for and professor

• Professors
  - office hours, teaching courses, TA for each one
Students, TAs and Professors (cont)

- The goal is not only to correctly implement but to also capture each concept separately.
  - design is equally important
Students, TAs and Professors (cont)

- Doing this only with Classes and inheritance
  - TA to inherit from Student and Prof,
    - impossible in Java
  - Can make Prof and Student inherit from TA
    - exposes unused methods inside Prof and TA
  - Use inheritance for construction
    - keep inside TA an instance of Student and Prof
      - lose substitutability (less flexible design)

- Let's try interfaces
  - define a interface for each role (TA, Prof, Student) that enforces each role's features
Students, TAs and Professors (cont)
Students, TAs and Professors (cont)

- Declare a type `Personify` as a Java interface type holding features shared by all.
- Declare an abstract class `Person` that implements these common features.
- For each role, define a corresponding interface:
  - abstract away common behavior (i.e. `Teaching`)
- Create implementation classes for each of the roles:
  - a student `implements` the Student Interface
  - a TA `implements` the TA Interface
  - a Professor `implements` the Professor Interface
- Check the source code on the class web page.
Dynamic Data Structures

• Data Structures that have the ability to dynamically alter some of their properties like
  - e.g. size

• Some examples
  - LinkedList, Queues, Trees, HashTables
  - standard implementations are available in the standard Java library classes. Most of them under `java.util`

• We will examine some of these
LinkedList

- a collection of locations with references from one cell to the next
- SingleLinkedList

![SingleLinkedList Diagram]

DoublyLinkedList

![DoublyLinkedList Diagram]
LinkedList (Java)

- Rich set of operations
  - add
    - at a specific index, begging, end
  - size
  - remove
    - an Object, first, last, at a specific index
- Return methods give you back an instance of type Object
LinkedList (Java) and Iterators

- Java provides a convenient way to go through a list, an iterator
- `iterator()` - returns an instance of an iterator initialized to point to the first element of the list.
- Iterators can alter the underlying list elements!

```java
LinkedList myList = new LinkedList();
myList.add("The");
myList.add("quick");
myList.add("brown");
myList.add("fox");
Iterator it = myList.iterator();
while(it.hasNext()){
    System.out.print((String)it.next());
    System.out.print(" ");
}
System.out.println("!");
```
Stack

- LIFO stack of objects
- Operations
  - push(Obj) – place something on the top of the stack

push(a);
push(b);
push(c);
push(d);
Stack (cont)

- LIFO stack of objects
- Operations
  - push(Object) – place something on the top of the stack
  - pop():Object – remove the first element of the stack

```plaintext
pop();
```
Stack (cont)

- LIFO stack of objects
- Operations
  - push(Object) – place something on the top of the stack
  - pop():Object – remove the first element of the stack
  - peek():Object – look at the first element without removing it

peek();
Stack (cont)

- LIFO stack of objects
- Operations
  - `push(Object)` – place something on the top of the stack
  - `pop():Object` – remove the first element of the stack
  - `peek():Object` – look at the first element without removing it
  - `empty():boolean` – check to see if the stack is empty